Parallel I/O for climate models

V. Balaji
SGI/GFDL

Fifth European SGI/Cray MPP Workshop
Bologna, September 1999

Parallel I/O

“I/O certainly has been lagging in the last decade.” – Seymour Cray, Public Lecture (1976).

“Also, I/O needs a lot of work.” – David Kuck, Keynote Address, 15th Annual Symposium on Computer Architecture (1988).

“I/O has been the orphan of computer architecture.” – Hennessy and Patterson, Computer Architecture - A Quantitative Approach. 2nd Ed. (1996).

Parallel I/O

- Dataset distributed across many processors, to be written to a single file.

- Single file distributed across multiple physical disks and I/O channels.

In the MPP context, we are principally concerned with the first of these, though here we shall allow the second as well.

Requirements

- Compact dataset (no descriptor files, etc).
- Final dataset to bear no trace of parallelism.
- Insulated from underlying APIs, which must nonetheless be accessible.
- Modularity, extensibility, support for specific data formats (esp. netCDF).
- F90 bindings.

I/O for climate models

Climate and weather models typically write a lot more data than they read. Reading is typically done once per run (input and restart files), whereas writing is done frequently (3D snapshots written to disk at regular intervals).

Restart files are written at machine precision. Snapshots are generally saved at 32 bits. (netCDF uses IEEE 32 bit data).

Climate experiments are often done in ensemble, and data shared among many institutions.

GFDL climate models

GFDL climate models are mostly gridpoint finite-difference models, on logically rectilinear grids. There is a strong emphasis on F90. Scientists tend to do their own programming. Parallelism is based on the distributed memory, message passing model.

- MOM: Modular Ocean Model.
- GFDL Hurricane model.
- New (as yet unnamed) flexible modeling interface: includes spectral and gridpoint hydrostatic atmospheric dynamical cores, NH dynamical core under development, land, ice, ocean model interfaces, coupler.

Parallel programming interface

GFDL has a homegrown parallelism API written as a set of 3 F90 modules:

- `mpp_mod` is a low-level interface to message-passing APIs (currently SHMEM and MPI; MPI-2 and Co-Array Fortran to come);
- `mpp_domains_mod` is a set of higher-level routines for domain decomposition and domain updates;
- `mpp_io_mod` is a set of routines for parallel I/O.

`http://www.gfdl.gov/~vb`

mpp_mod

mpp_mod is a set of simple calls to provide a uniform interface to different message-passing libraries. It currently can be implemented either in the SGI/Cray native SHMEM library or in the MPI standard. Other libraries (e.g MPI-2, Co-Array Fortran) can be incorporated as the need arises.

mpp_mod is currently in use in all models at GFDL.

mpp_mod design issues

- Simple, minimal API, with free access to underlying API for more complicated stuff.

- Design toward typical use in climate/weather CFD codes (rectilinear grid, halo update, data transpose).

- Performance to be not significantly lower than any native API.

The module is coded in F90 and makes extensive use of certain f90 features that greatly enhance simplicity without affecting performance (e.g function overloading). Where performance is suspect, some f77-style features are used (e.g pass-by-address to `mpp_transmit`). It uses one non-standard yet widely used feature, *Cray pointers*, which are functionally equivalent to C pointers, and are fairly widely available even on non-SGI platforms.

mpp_mod API

- Basic calls:
  - `mpp_init()`
  - `mpp_exit()`
  - `mpp_transmit()`: basic message passing call. Typical use assumes *two* transmissions per domain, e.g halo update.
  - `mpp_sync()`
- Reduction operators:
  - `mpp_max()`
  - `mpp_sum()`: provides bit-reproducible $\log_2 p$ algorithm as option.

mpp_transmit performance

SHMEM implementation of mpp_transmit on T3E:

- Latency: 11 μs (3 μs for bare `shmem_get`).
- Peak bandwidth: 300 Mb/s.
- For messages longer than 1000 words, the two are not distinguishable.

Latency increase is due to code to handle dynamic arrays.

MPI bandwidth is 150 Mb/s. T90 bandwidth is 5 Gb/s.

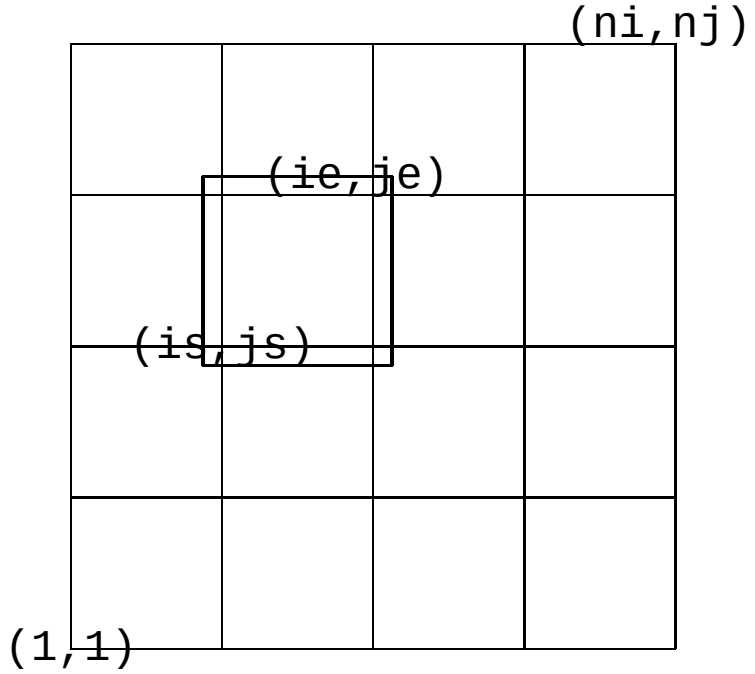
mpp_domains_mod

Comprehensive domain decomposition information is held in a derived type *domaintype*. We define *domain* as the grid associated with a *task*. We define the *compute domain* as the set of gridpoints that are computed by a task, and the *data domain* as the set of points that are required by the task for the calculation. There can in general be more than 1 task per PE, though often the number of domains is the same as the processor count. We define the *global domain* as the global computational domain of the entire model (i.e, the same as the computational domain if run on a single processor).

2D domains are defined using a derived type `domain2D`, constructed as follows (see comments in code for more details):

```
type, public :: domain_axis_spec
   sequence
   integer :: start_index, end_index, size, max_size
   logical :: is_global
end type domain_axis_spec
type, public :: domain1D
   sequence
   type(domain_axis_spec) :: compute, data, global
   integer :: ndomains
   integer :: pe
   integer, dimension(:), pointer :: pelist
   type(domain1D), pointer :: prev, next
end type domain1D
```

```
!domaintypes of higher rank can be constructed from type domain1D
  type, public :: domain2D
     sequence
     type(domain1D) :: x
     type(domain1D) :: y
     integer :: pe
     type(domain2D), pointer :: west, east, south, north
  end type domain2D
```



The `domain2D` type contains all the necessary information to define the global, compute and data domains of each task, as well as the PE associated with the task. The PEs from which remote data may be acquired to update the data domain are also contained in a linked list of neighbours.

mpp_domains_mod calls:

- mpp_define_domains()
- mpp_update_domains()

```
type(domain2D) :: domain(0:npes-1)
call mpp_define_domains( (/1,ni,1,nj/), domain, xhalo=2, yhalo=2 )
...
!allocate f(i,j) on data domain
!compute f(i,j) on compute domain
...
call mpp_update_domains( f, domain(pe) )
```

mpp_io_mod: a parallel I/O interface

mpp_io_mod is a set of simple calls to simplify I/O from a parallel processing environment. It uses the domain decomposition and message passing features of mpp_mod and mpp_domains_mod . It is designed to deliver high-performance I/O from distributed data, in the form of self-describing files (verbose metadata).

mpp_io_mod features

- Simple, minimal API, with freedom of access to native APIs.
- Strong focus on performance of parallel write.
- Accepts netCDF format, widely used in the climate/weather community.
- May require post-processing, generic tool for this to be provided by GFDL.

mpp_io_mod output modes

`mpp_io_mod` supports three types of parallel I/O:

- Single-threaded I/O: a single PE acquires all the data and writes it out.
- Multi-threaded, single-fileset I/O: many PEs write to a single file.
- Multi-threaded, multi-fileset I/O: many PEs write to independent files (requires post-processing).

mpp_io_mod API

- `mpp_io_init()`
- `mpp_open()`
- `mpp_close()`
- `mpp_read()`
- `mpp_write()`
- `mpp_write_meta()`

Metadata

Since the datasets are required to be compact (comprehensively self-describing) we associate metadata in the file header associated with each axis and field in the file. Metadata contains names and units for each variable, as well as associating each field with a number of axes. Optional attributes can be specified to describe data masks, missing data, scaling, packing, etc. These use the derived types `axistype` and `fieldtype` use associated from `mpp_io_mod` .

mpp_open

The key call is `mpp_open()`. Most information about type of I/O to be performed is contained here:

```
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32,
access=MPP_SEQUENTIAL, threading=MPP_SINGLE )
```

Format can be one of `MPP_ASCII`, `MPP_IEEE32`, `MPP_NATIVE`, or `MPP_NETCDF`.

Single-threaded I/O from multiple PEs means PE0 will acquire all the data and do the actual write.

Multi-threaded I/O

```
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32,
access=MPP_SEQUENTIAL, threading=MPP_MULTI, fileset=MPP_MULTI
)
```

Multi-threaded I/O can have all PEs write to a single file or each to an independent file, which must later be assembled (a generic tool for this is available). It offers the possibility of high-performance I/O when parallel filesystems are buggy or slow.

mpp_io_mod calling sequence

```
type(domain2D) :: domain(0:npes-1)
type(axistype) :: x, y, z, t
type(fieldtype) :: field
integer :: unit
character*(*) :: file
real, allocatable :: f(:,:,:)
call mpp_define_domains( (/1,ni,1,nj/), domain )
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32, &
  access=MPP\_SEQUENTIAL, threading=MPP_SINGLE )
call mpp_set_filespec( unit, '-F cachea' )
call mpp_write_meta( unit, x, 'X', 'km', ... )
...
call mpp_write_meta( unit, field, (/x,y,z,t/), 'Temperature', 'kelvin', ... )
...
call mpp_write( unit, field, domain(pe), f, tstamp )
```

mpp_io_mod performance on T90/T3E

Multi-threaded I/O offers a simple way to stripe the data across as many I/O channels and disk channels as are available, using an appropriate combination of `mpp_open` specs and *assign*.

On GFDL systems, we have DD-302 disks with a peak I/O rate of 10Mb/s available on both the T90 and the T3E, and DA-302 disk arrays with a peak I/O rate of 40 Mb/s for large well-formed I/O on T90. T3E I/O is mediated by the O/S PEs, allowing two channels.

`mpp_io_mod` shows linear scaling on DD-302s up to 4 PEs on T90, but is limited on the T3E to 20Mb/s. On the T90 4-way parallel writes to DA302 for multifile I/O has been measured up to 160 Mb/s from 4 PEs to 4 files.

Conclusions

- `mpp_io_mod` provides a very simple set of calls for writing compact datasets from distributed data. It has proved versatile and easy to use, and currently has become the labwide basis for all I/O in GFDL models.

- It provides access to underlying I/O environments (such as `assign`) and is easily extensible to any I/O API. It currently provides standard fortran and netCDF I/O.

- It is publicly available through my GFDL webpage:

  `http://www.gfdl.gov/~vb`